Algorithms PA 2 Report

B11901123 電機三 張牲源

1. Data Structure

1) Edge Endpoint Storage

vector<int> edgeEndpoints; // Stores the endpoint of each vertex

a. Structure: One-dimensional vector where the index represents the vertex.

b. Chord Representation: For a chord between vertices i and j:

- edgeEndpoints[i] = j
- edgeEndpoints[j] = i

c. Initial State: Value -1 indicates no connected chord.

d. Space Complexity: O(n), where n is the number of vertices.

e. Advantages:

- O(1) lookup vs. O(logn) for map/set.
- Cache-friendly due to contiguous memory.

2) Dynamic Programming Table

vector<vector<int>> memoTable; // Dynamic programming memoization table

a. Triangular Structure Optimization:

```
void setupMemoization(int size) {
    memoTable.resize(size);
    for (int i = 0; i < size; i++) {
        memoTable[i].assign(size - i, -1);
    }
}
```

b. Access Pattern: *memoTable[left][right-left]* stores results for vertices from left to right.

c. Uncomputed State: -1 indicates an uncomputed subproblem.

d. Space Optimization:

- Traditional matrix: $n \times n = n^2$ entries.
- Triangular structure: $\approx \frac{n^2}{2}$ entries.
- Reduces space usage by ~50%.

3) Helper Functions for Table Access

```
int getMemoValue(int left, int right) {
   return (right <= left) ? 0 : memoTable[left][right - left];
}
```

- Purpose: Encapsulates boundary checking.
- Benefits: Ensures consistent access patterns and prevents out-of-bounds errors.

2. Algorithm

The program uses a top-down dynamic programming approach to solve the Maximum Planar Subset problem, which aims to find the maximum number of non-overlapping chords in a planar graph. Each chord is represented as a pair of vertices. The algorithm finds both the number of these non-crossing chords and the specific pairs of vertices that make up this subset.

The main approach involves:

1. Storing the endpoints of each vertex in a one-dimensional array.
2. Using a memoization table to store results of subproblems, avoiding redundant computations.
3. Applying a recursive function with memoization to count the maximum number of non-crossing chords within a given range of vertices.
4. Reconstructing the solution to list the specific pairs of chords in the maximum planar subset.

1) **Dynamic Programming Recurrence (findMaxNonCrossingEdges)** The findMaxNonCrossingEdges function calculates the maximum number of non-crossing chords for any interval [left, right] using a recursive formula.

   a. **Base Case**: If right <= left, the interval is invalid or has no vertices between left and right, so no chords can exist. Thus:

      if (right <= left) return 0;

   b. **Recursive Cases**:

- **Direct Connection**: If there is a chord directly connecting left and right (i.e., *edgeEndpoints[left] == right*), the solution includes this chord. The recurrence is: $M(left,right) = 1+M(left+1,right-1)$

  This means that we include the chord (left, right) and recursively calculate the maximum non-crossing chords for the interval [left + 1, right - 1].

- **Intermediate Connection**: If right is connected to an intermediate vertex k within [left, right] (i.e., *edgeEndpoints[right] > left && edgeEndpoints[right] < right*), we have two choices:
    - **Include the chord** between k and right, splitting the interval into two subproblems: $M(left,right)=1+M(left,k-1)+M(k+1,right-1)$
    - **Exclude the chord** and solve for the interval [left, right - 1]: $M(left,right) = M(left,right-1)$

  The result for M(left, right) is the maximum of these two values.

c. Memoization Check: Each result is stored in memoTable to avoid redundant calculations. If *memoTable[left][right - left]* already contains a value (i.e., not -1), it returns the stored value directly, reducing the time complexity by eliminating repeated computations.

```
int findMaxNonCrossingEdges(int left, int right) {
    if (right <= left) return 0;
    if (memoTable[left][right - left] != -1)
        return memoTable[left][right - left];
    int result;
    if (edgeEndpoints[left] == right) {
        result = 1 + findMaxNonCrossingEdges(left + 1, right - 1);
    }
    else if (edgeEndpoints[right] > left && edgeEndpoints[right] < right) {
        int withEdge = 1 + findMaxNonCrossingEdges(left, edgeEndpoints[right] - 1)
+ findMaxNonCrossingEdges(edgeEndpoints[right] + 1, right - 1);
        int withoutEdge = findMaxNonCrossingEdges(left, right - 1);
        result = max(withEdge, withoutEdge);
    }
    else {
        result = findMaxNonCrossingEdges(left, right - 1);
    }
```

```
    memoTable[left][right - left] = result;
    return result;
}
```

2) **Solution Reconstruction** After determining the maximum number of chords, the *reconstructSolution* function retrieves the specific chords in the maximum planar subset by following the choices made in the dynamic programming table:
    a. If there is a direct connection between left and right (i.e., *edgeEndpoints[left] == right*), the chord (left, right) is part of the solution.
    b. If right is connected to an intermediate vertex, the function checks if including or excluding this chord yields the optimal result. It reconstructs each subproblem based on the choices that maximize the result.

```
void reconstructSolution(ofstream& output, int left, int right) {
    if (right <= left) return;
    if (edgeEndpoints[left] == right) {
        output << left << " " << right << endl;
        reconstructSolution(output, left + 1, right - 1);
    }
    else if (edgeEndpoints[right] > left && edgeEndpoints[right] < right) {
        int withEdge = 1 + getMemoValue(left, edgeEndpoints[right] - 1) +
                    getMemoValue(edgeEndpoints[right] + 1, right - 1)
        if (getMemoValue(left, right) == withEdge) {
            reconstructSolution(output, left, edgeEndpoints[right] - 1);
            output << edgeEndpoints[right] << " " << right << endl;
            reconstructSolution(output, edgeEndpoints[right] + 1, right - 1);
        }
        else { reconstructSolution(output, left, right - 1);}
    }
    else {reconstructSolution(output, left, right - 1); }
}
```

3) **Main Execution (solve function)**

a. Reads input vertices and edges from the file and initializes the edgeEndpoints array to map each vertex's chord endpoint.

b. Calls setupMemoization to initialize the memoization table.

c. Executes *findMaxNonCrossingEdges(0, vertexCount - 1)* to compute the maximum number of non-crossing chords.

d. Calls reconstructSolution to list the specific chord pairs that make up the maximum planar subset, writing them to the output file.

```
void solve(const string& inputPath, const string& outputPath) {
    ifstream input(inputPath);
    if (!input) {
        throw runtime_error("Failed to open input file: " + inputPath);
    }
    ofstream output(outputPath);
    if (!output) {
        throw runtime_error("Failed to open output file: " + outputPath);
    }
    int vertexCount;
    input >> vertexCount;
    edgeEndpoints.assign(vertexCount, -1);
    int v1, v2;
    while (input >> v1 >> v2) {
        edgeEndpoints[v1] = v2;
        edgeEndpoints[v2] = v1;
    }
    setupMemoization(vertexCount);
    output << findMaxNonCrossingEdges(0, vertexCount - 1) << endl;
    reconstructSolution(output, 0, vertexCount - 1);
    input.close();
    output.close();
}
```

3. Time Complexity Analysis

1. Input Processing: $O(n)$
2. Memoization Table Setup: $O(n^2)$
3. Core Algorithm: $O(n^2)$
4. Solution Reconstruction: $O(k)$, the number of chords in the maximum planar subset, which can be up to $O(n)$ in the worst case.
5. Total= $O(n^2)$

Note: Space Complexity

- Static Storage: *edgeEndpoints* $O(n)$, *memoTable* $O(n^2)$.
- Recursive Stack: Max depth $O(n)$.

4. README

1) **Compile the Program**: In the root directory, type:

make

This command compiles the source code and creates the *mps* executable in the *bin/* directory.

2) **Run the Program**: Navigate to the bin/ directory and execute the program with:

./mps inputs/<input file name> outputs/<output file name>

Replace <*input file name*> with your input file containing vertices and chord endpoints, and <*output file name*> with the desired output file name.

5. Conclusion

In my program, I initially employed a bottom-up approach for solving the Maximum Planar Subset problem. I started by using two separate 2D arrays to store the subset cut and set size for each pair of vertices. However, this method was highly inefficient for large input sizes. When tested on an input case with n=100 , the program took approximately 40 minutes to complete due to the significant memory and computation overhead.

Realizing that only half of each 2D array was actually necessary for the calculations, I optimized the approach by merging the two arrays into a single 2D array. This change halved the memory

usage and improved access efficiency, reducing the runtime for the same 100,000 input case to around 20 minutes.

I also experimented with various sorting optimizations. Initially, I used a custom merge sort function implemented in a previous programming assignment. Then, I tested a randomized quick sort as well as the C++ standard library's *std::sort* function, which employs a combination of quicksort, heapsort, and insertion sort for optimal performance. Despite these attempts, none of these sorting methods provided a substantial runtime improvement.

Ultimately, the top-down approach emerged as the most efficient solution. By employing memoization, this approach computes only the necessary subproblems, significantly reducing the computational workload compared to the bottom-up approach. Additionally, the top-down method leverages recursive calls to naturally align with the problem structure, minimizing unnecessary table fills and maximizing cache efficiency. As a result, it not only reduced the computation time for large inputs but also made the code more maintainable and straightforward to debug.